

Avoiding Contradictions in the Paradoxes, the Halting Problem, and Diagonalization

Timothy J. Armstrong
t.armstrong888@gmail.com

Abstract

The fundamental proposal in this article is that logical formulas of the form $(f \leftrightarrow \neg f)$ are not contradictions, and that formulas of the form $(t \leftrightarrow t)$ are not tautologies. Such formulas, wherever they appear in mathematics, are instead reason to conclude that f and t have a third truth value, different from true and false. These formulas are circular definitions of f and t . We can interpret the implication formula $(f \leftrightarrow \neg f)$ as a rule, a procedure, to find the truth value of f on the left side: we just need to find the truth value of f on the right side. When we use the rules to ask if f and t are true or false, we need to keep asking if they are true or false over and over, forever.

Russell's paradox and the liar paradox have the form $(f \leftrightarrow \neg f)$. The truth value provides a straightforward means of avoiding contradictions in these problems. One broad consequence is that the technique of proof by contradiction involving formulas of the form $(f \leftrightarrow \neg f)$ becomes invalid. One such proof by contradiction is one form of proof that the halting problem is uncomputable. The truth value also appears in Cantor's diagonal argument, Berry's paradox, and the Grelling-Nelson paradox.

1 Introduction

Consider these Prolog rules¹:

```
t :- t.
f :- \+ f.
a(X) :- a(X).
b(X) :- \+ b(X).
elementOf(X, c) :- elementOf(X, X).
elementOf(X, r) :- \+ elementOf(X, X).
```

The roughly corresponding logical formulas are:

$$t \leftrightarrow t \quad (1)$$

$$f \leftrightarrow \neg f \quad (2)$$

$$\forall x(A(x) \leftrightarrow A(x)) \quad (3)$$

$$\forall x(B(x) \leftrightarrow \neg B(x)) \quad (4)$$

$$\forall x(x \in C \leftrightarrow x \in x) \quad (5)$$

$$\forall x(x \in R \leftrightarrow x \notin x) \quad (6)$$

The Prolog predicate “`elementOf`” is meant to be the standard “ \in ” symbol in set theory, and the rules involving “`elementOf`” are meant to represent Russell's paradox.

People familiar with Prolog should recognize that the program enters infinite recursion when we run these queries on the command line, with any constant “`z`”:

```
?- t.
?- f.
?- a(z).
?- b(z).
?- elementOf(c, c).
?- elementOf(r, r).
```

When we use Prolog to ask if the statements `t`, `f`, `a(z)`, `b(z)`, `elementOf(c, c)`, and

¹For an introduction to Prolog, a logic programming language, see Clocksin and Mellish [4].

`elementOf(r, r)` are true or false, we keep asking if they are true or false over and over, infinitely, and never arrive at an answer of true or false. As part of the procedure to find the truth value of each statement, we need to find the truth value of the same statement.

We should call this behavior a “truth value”. Some statements are true, other statements are false, and still other statements have the behavior that when we ask if they are true or false, we keep asking forever. This sort of infinite recursion is familiar in Prolog, but we need to account for it in all forms of logic.

We would ideally like Prolog to return an answer of “recursive” instead of “true” or “false”. Detecting infinite recursion in general is the halting problem, but people have successfully developed algorithms to detect infinite recursion in special cases, as in the field of termination analysis [1]. There could be an option for attempting to detect infinite recursion when running a Prolog program, if it would be too computationally expensive to check for infinite recursion all the time.

This truth value is important because it appears in Russell’s paradox, the liar paradox, the halting problem, Cantor’s diagonal argument, Berry’s paradox, and the Grelling-Nelson paradox. Many of these problems involve formulas of the form $(f \leftrightarrow \neg f)$. People conventionally take these formulas to be contradictions. What Prolog’s particular resolution-based theorem proving algorithm says about the statements in these problems and the above Prolog statements is that they are not true, are not false, and are not both true and false at the same time; they are not contradictions. We should treat these statements as having the recursive truth value in all forms of logic. We need to develop a three-valued logic for this truth value; Fitting [6] provides some of what is needed.

The rest of this article is organized as follows. First it is presented how Russell’s paradox has the recursive truth value. Next, it is presented generically how $(t \leftrightarrow t)$ and $(f \leftrightarrow \neg f)$ are recursive instead of being a tautology and a contradiction. Afterwards, Tarski’s Convention T is used to prove that the liar paradox is recursive. The truth value has consequences for the technique of proof by contradiction, and one proof

by contradiction that the halting problem is uncomputable is analyzed. Finally, Cantor’s diagonal argument is presented briefly, as well as how some numbers have this truth value in the place of some digits. Berry’s paradox and the Grelling-Nelson paradox [8, intro] are left for presentation elsewhere.

2 Russell’s Paradox

Russell’s paradox [9] involves the set:

$$R = \{x \mid x \notin x\} \quad (7)$$

R is the set of everything that is not a member of itself. C is the set of everything that *is* a member of itself:

$$C = \{x \mid x \in x\} \quad (8)$$

The above rules for R and C are repeated here:

```
elementOf(X, c) :- elementOf(X, X).
elementOf(X, r) :- \+ elementOf(X, X).
```

```
\forall x(x \in C \leftrightarrow x \in x)
\forall x(x \in R \leftrightarrow x \notin x)
```

For any x , x is an element of R if and only if x is not an element of x . We ask if $R \in R$ is true or false:

```
?- elementOf(r, r).
```

Prolog enters into infinite recursion. In the Prolog program, we provide a *procedure* for determining if an arbitrary entity x is an element of R . In order to find out if x is an element of R , we need to find out if x is an element of itself. In order to find out if R is an element of R , we need to find out if R is an element of itself.

It is in general desirable for sets to have a decidable procedure to determine if any entity is an element of the set. For some sets, we can write computer programs to decide membership, as in logic programming languages like Prolog or in imperative programming languages. We can provide logical rules so that we can use theorem proving techniques to decide membership. It happens that in Prolog’s particular theorem proving algorithm, the equivalent of

$\forall x(x \in R \leftrightarrow x \notin R)$ becomes infinitely recursive when we use the rule to ask if $R \in R$. Prolog interprets the implication formula as a *procedure* to determine if $R \in R$:

$$R \in R \leftrightarrow R \notin R \quad (9)$$

If we can determine that $R \notin R$, we can conclude that $R \in R$.

Russell's paradox is strange when we describe it informally: We ask if R is an element of R . R is an element of R if and only if R is not an element of R . In other words, R is an element of R if it holds that R is not an element of R , R is an element of R if it is the case that R is not an element of R , and R is an element of R on the condition that R is not an element of R . That means we have to ask: is R an element of R ? Asking if R is an element of R is what we were doing at the beginning, so we ask again. We repeat the process of asking if R is an element of R . When we ask if $R \in R$ is true or false, we keep asking if $R \in R$ is true or false over and over, forever.

Is $R \in R$ *actually* true or false, just we do not know? Prolog's theorem proving algorithm leads us to conclude that $R \in R$ is neither true nor false. Instead, it has a different truth value than true or false, the recursive truth value. Is R *actually* either in R or not in R , just we do not know? We can never say that R is either in the set or not in the set. Instead, we keep asking forever when we ask if it is in the set. An entity may be related to a set in a manner other than being an element of it or not an element of it.

3 Tautologies and Contradictions

Russell's paradox has the form of the propositional logic formula $(f \leftrightarrow \neg f)$:

$$(R \in R) \leftrightarrow \neg(R \in R) \quad (10)$$

$$\text{elementOf}(r, r) \leftrightarrow \neg \text{elementOf}(r, r) \quad (11)$$

The liar paradox, presented in section 4, also has the

form $(f \leftrightarrow \neg f)$: $(\text{True}(s) \leftrightarrow \neg \text{True}(s))$. The corresponding Prolog rule is:

$$f :- \neg f.$$

f is true if and only if f is false. f is true if it holds that f is false; f is true on the condition that f is false. The Prolog rule provides a means, a procedure, for finding the truth value of f . In general, if we want to find the truth value of f , we need to search the Prolog database to find if there is a fact asserting f , or if there is a rule with f as its head. We find the rule “ $f :- \neg f$.”, and we attempt to satisfy the body of the rule. As part of the procedure to find the truth value of f , we need to find the truth value of f . When we ask if f is true or false, we keep asking repeatedly forever.

In classical two-valued logic, we often interpret an implication statement $(p \leftrightarrow q)$ as providing a procedure, a rule, for finding the truth value of p : we just need to find the truth value of q . This interpretation of implication is explicit in Prolog. We should interpret $(f \leftrightarrow \neg f)$ as providing a rule for finding the truth value of f on the left side: we just need to find the truth value of f on the right side.

If we somehow know that f is either true or false, the formula $(f \leftrightarrow \neg f)$ would force us to conclude that f has the opposite truth value, which would be a contradiction. However, if all we have is the formula $(f \leftrightarrow \neg f)$, it is just a rule for finding the truth value of f : an infinitely recursive rule. f is true if and only if f is false. So, using this rule, in order to find out if f is true, we need to find out if f is false.

Also consider the formula $(t \leftrightarrow t)$ and the Prolog rule “ $t :- t$.”. As part of the procedure to find the truth value of t , we need to find the truth value of t . The set C above has the form $(t \leftrightarrow t)$:

$$(C \in C) \leftrightarrow (C \in C) \quad (12)$$

$$\text{elementOf}(c, c) \leftrightarrow \text{elementOf}(c, c) \quad (13)$$

As part of the procedure to find out if C is an element of C , we need to find out if C is an element of C .

In classical two-valued logic, $(f \leftrightarrow \neg f)$ is a contradiction, and $(t \leftrightarrow t)$ is a tautology. The proposal in this article is that we should instead treat these

formulas as reason to conclude that f and t have the recursive truth value. f and t have just a *single* truth value; they are not true, are not false, and are not both true and false at the same time. We should treat all formulas with the *form* $(f \leftrightarrow \neg f)$ or $(t \leftrightarrow t)$ as being infinitely recursive, such as Russell's paradox and the liar paradox.

It makes sense to say that $(t \leftrightarrow t)$ is a tautology in that, if we know that t is true, we can conclude that t is true. On the other hand, if we intend $(t \leftrightarrow t)$ to be a *rule* for finding the truth value of t , as in Prolog, then we would say that $(t \leftrightarrow t)$ is not a tautology but instead says something special about the truth value of t , that t has the recursive truth value.

Saying that $(f \leftrightarrow \neg f)$ is not a contradiction seems like a bold claim. For one matter, it would invalidate one form of the technique of proof by contradiction. We should say that it is still a contradiction if a statement has more than one truth value at the same time, such as $(p \wedge \neg p)$; proofs by contradiction of that sort would still be valid. However, proofs by contradiction that depend on a formula of the form $(f \leftrightarrow \neg f)$ being a contradiction would be invalid. We would need to sort through all of mathematics to find all the proofs by contradiction that have this form, and figure out how to correct the proofs and all the theory built on top of those proofs. It would be a very large task.

Formulas of the form $(f \leftrightarrow \neg f)$ would be legitimate to have as axioms or theorems in a formal theory, or as data in a knowledge base, and would not make the theory or knowledge base inconsistent. That observation is consequential for the paradoxes. Formulas of the form $(t \leftrightarrow t)$ would not be *harmless* to have in a theory or knowledge base.

4 The Liar Paradox

It may be evident from what was presented above that the liar paradox has the recursive truth value. There is much more that needs to be said about the liar paradox, which is not included in this article for space considerations. Let us briefly consider, though, Tarski's well-known "Convention T" [10]. He writes:

Let us consider an arbitrary sentence; we shall replace it by the letter ' p .' We form

the name of this sentence and we replace it by another letter, say ' X .' We ask now what is the logical relation between the two sentences " X is true" and ' p .' It is clear that from the point of view of our basic conception of truth these sentences are equivalent. In other words, the following equivalence holds:

(T) X is true if, and only if, p .

Tarski provides the example of the sentence "snow is white":

The sentence "snow is white" is true if, and only if, snow is white.

For the liar paradox, let us represent with the letter ' s ' the sentence "This sentence is not true", or equivalently, "Sentence ' s ' is not true". Then, by Convention T: "Sentence ' s ' is true if, and only if, sentence ' s ' is not true." We can formalize the formula roughly as:

$$\text{True}(s) \leftrightarrow \neg \text{True}(s) \quad (14)$$

The formula has the form $(f \leftrightarrow \neg f)$. "**True**" is the truth predicate that asserts that the argument is a true sentence. What we want to say in connection to the recursive truth value is that Convention T provides a bidirectional rule: if we know that p , we can conclude that X is true; if we know that X is true, we can conclude that p . The most *direct* way to find out if X is true is to find out if p . The most direct way to find out if the sentence "snow is white" is true is to find out if snow is white. The most direct way to find out if sentence ' s ' is true is to find out if ' s ' is not true.

Convention T provides a *procedure* for finding out if X is true: we need to find out if p . Using this rule, in order to find out if ' s ' is true, we need to find out if ' s ' is true. In order to find the truth value of ' s ', we need to find out if its claim about reality is correct.

' s ' thus has the recursive truth value. ' s ' is true if it is not true and is not true if it is true; so, when we ask if ' s ' is true, we need to ask again repeatedly forever. We initially do not know the truth value

of ‘ s ’. We need some means, some procedure, for finding its truth value. Convention T provides such a procedure.

It is similar for “This sentence is true”, which is true if and only if it is true. If we call that sentence ‘ u ’, we can write:

$$\text{True}(u) \leftrightarrow \text{True}(u) \quad (15)$$

which has the form $(t \leftrightarrow t)$. As part of the process to find out if ‘ u ’ is true, we need to find out if ‘ u ’ is true. (We would need to consider in more detail elsewhere the difference between ‘ s ’ and “This sentence is false”, but in any case both are infinitely recursive.)

5 The Halting Problem as a Proof by Contradiction

One notable proof by contradiction with the form $(f \leftrightarrow \neg f)$ is the proof by Davis et al. that the halting problem is uncomputable [5, ch 4]. Turing’s proof is a bit different, but Davis et al.’s proof is explicitly in this form. The authors assume that the halting problem is computable and then arrive at this formula that they claim is a contradiction:

$$\text{HALT}(y_0, y_0) \leftrightarrow \neg \text{HALT}(y_0, y_0) \quad (16)$$

They conclude, by a proof by contradiction, that their assumption that halting problem is computable must be false.

Equation 16 has the form $(f \leftrightarrow \neg f)$. It makes sense, given how the authors present the halting problem, to say that $\text{HALT}(y_0, y_0)$ has the recursive truth value: to say that, in order to find the truth value of $\text{HALT}(y_0, y_0)$ on the left side, we need to find the truth value of $\text{HALT}(y_0, y_0)$ on the right side.

It is worth examining in a bit more detail. The authors discuss computability mainly using an imperative programming language they devised, instead of using Turing machines. $\text{HALT}()$ is a computer program that takes as its first argument a natural number x , and as its second argument a natural number y representing an arbitrary computer program, and is supposed to decide if the program y running with the

```
unsigned int P(unsigned int x)
{
    A: if (HALT(x, x)) goto A;

    return 0;
}
```

Listing 1: The program \mathcal{P} that is problematic for the halting problem, given in C/C++ syntax. To explain the code for readers unfamiliar with C/C++: The first “`unsigned int`” means that the return value of the program is a natural number. The second “`unsigned int`” means that the program takes a single parameter “`x`” that is a natural number. “`A:`” is a label for the given line. If the $\text{HALT}(x, x)$ procedure call evaluates to “`true`”, the “`goto A`” command causes the program to enter into an infinite loop, repeatedly executing line A. If the $\text{HALT}(x, x)$ procedure call evaluates to “`false`”, the program reaches the “`return 0;`” command, which halts the program and returns the answer of 0.

input of x would either halt or run forever (on an idealized computer). $\text{HALT}()$ provides a return value of “`true`” (or “1”) if program y would halt and “`false`” (or “0”) if it would not halt. $\text{HALT}()$ is supposed to compute the function:

$$\text{HALT}(x, y) = \begin{cases} 1 & \text{if program } y \text{ running} \\ & \text{with input } x \text{ halts} \\ 0 & \text{otherwise} \end{cases} \quad (17)$$

The authors construct a certain program \mathcal{P} that is problematic:

[A] IF HALT(X, X) GOTO A

\mathcal{P} translated into C/C++ syntax² (with which the reader may be more familiar) is in listing 1. y_0 is the natural number that represents \mathcal{P} .

Consider what happens when we run \mathcal{P} with the input of y_0 , that is when we run $\mathcal{P}(y_0)$. If, inside

²For an introduction to C, an imperative programming language, see Kernighan and Ritchie [7].

\mathcal{P} , the $\text{HALT}(y_0, y_0)$ procedure call returns “true” (saying $\mathcal{P}(y_0)$ halts), then $\mathcal{P}(y_0)$ does not halt. If the $\text{HALT}(y_0, y_0)$ procedure call returns “false” (saying $\mathcal{P}(y_0)$ does not halt), then $\mathcal{P}(y_0)$ halts. Thus, as in equation 16:

$$\text{HALT}(y_0, y_0) \leftrightarrow \neg \text{HALT}(y_0, y_0)$$

What we want to say here about the recursive truth value is as follows. In order to find out if $\mathcal{P}(y_0)$ halts (that is, in order to find the truth value of $\text{HALT}(y_0, y_0)$), we need to find the return value of the $\text{HALT}(y_0, y_0)$ procedure call inside \mathcal{P} . That is, in order for us to find the truth value of $\text{HALT}(y_0, y_0)$, we need the $\text{HALT}()$ program to tell us the truth value of $\text{HALT}(y_0, y_0)$.

In this interpretation, $\text{HALT}(y_0, y_0)$ has the recursive truth value. When we ask if $\text{HALT}(y_0, y_0)$ is true or false, we – or the $\text{HALT}()$ program – need to keep asking if $\text{HALT}(y_0, y_0)$ is true or false over and over, forever.

It is simplest to say that Davis et al.’s proof by contradiction, asserting that the halting problem is uncomputable, is invalid because equation 16, having the form $(f \leftrightarrow \neg f)$, is not actually a contradiction.

6 Diagonalization and the Halting Problem

The reader may be able to imagine how the recursive truth value relates to Cantor’s diagonal argument [3] and to Turing’s original article on his version of the halting problem [11]. These topics require more extended presentation, but we should say a few words briefly here.

Some numbers have the recursive truth value in the place of some digits. Say we have written a computer program to perform the computation of finding the digits of a real number. For some numbers and for some digits, when we attempt to find the value of the digit, we need to attempt again to find the value of the same digit. The program enters infinite recursion. When we ask what the value of the digit is, we keep asking what the value is over and over, forever. For

one example, if we interpret the set in Russell’s paradox as a real number, it has an infinitely recursive digit.

If it is possible to detect infinite recursion, though, the program can just mark the given digit as having the recursive truth value, such as with an “r”, and move on to computing the next digit. For example, we could write: “0.10r0110...”

There is a number with an infinitely recursive digit in Turing’s article in section 8, “Application of the diagonal process”, which is the key section for the halting problem and the recursive truth value. In Turing’s article, in order to find the $R(K)$ -th digit of β' , the machine needs to find the $R(K)$ -th digit of β' . However, the recursive truth value allows us to handle such a number. What the machine can do is simply mark the $R(K)$ -th digit as having the recursive truth value, such as by printing an “r” on the tape, and move on to computing the next digit in β' . In this way, the machine running with its own program number as input becomes less problematic.

For Cantor’s diagonal argument, it happens that, when we attempt to include the diagonal and anti-diagonal real numbers as rows in the matrix, the numbers acquire an infinitely recursive digit on the diagonal. A proof requires more extended presentation, but let us just comment on a mathematical equation that Boolos et al. use to explain diagonalization [2, ch 2]. They assert that this equation is a contradiction:

$$s_m(m) = 1 - s_m(m) \quad (18)$$

$s_m(m)$ is supposed to take the value of either 0 or 1. We should treat this equation as being infinitely recursive instead of as a contradiction: in order to find the value of $s_m(m)$ on the left, we need to find the value of $s_m(m)$ on the right. We define the value of $s_m(m)$ to be 1 if the value of $s_m(m)$ is 0, and to be 0 if the value of $s_m(m)$ is 1.

There is a similar equation in Turing’s article in section 8, which we can rearrange to:

$$\phi_K(K) = 1 - \phi_K(K) \quad (19)$$

Turing asserts that this equation is a contradiction, but we should instead treat it as infinitely recursive:

in order to find the value of $\phi_K(K)$ on the left, we need to find the value of $\phi_K(K)$ on the right.

7 Conclusion

The reader should hopefully find it plausible, and perhaps convincing, that it is best to treat $(f \leftrightarrow \neg f)$ as infinitely recursive instead of as a contradiction. It seems very clear in Prolog that $(t \leftrightarrow t)$ and $(f \leftrightarrow \neg f)$ lead us to conclude that t and f have the recursive truth value. It provides a convenient means of avoiding contradictions in the paradoxes, the halting problem, and diagonalization. This approach to handling the paradoxes provides an alternative to Zermelo-Fraenkel set theory, type theory, and Tarski's hierarchy of languages. We would need to re-work the foundations of logic and mathematics to include this truth value.

We can consider how the truth value works in propositional logic with $(t \leftrightarrow t)$ and $(f \leftrightarrow \neg f)$, before considering how it works in first-order logic and other forms of logic. It would be necessary to figure out for all the proof systems (truth tables, resolution, tableaux, axiomatic systems, etc.) how to prevent them from proving that $(t \leftrightarrow t)$ is a tautology, and how to prevent them from proving that $(f \leftrightarrow \neg f)$ is a contradiction. It would be necessary to figure out how to adapt the proof systems so that, when given $(t \leftrightarrow t)$ and $(f \leftrightarrow \neg f)$ as premises, they prove that t and f have the recursive truth value.

References

- [1] *14th International Workshop on Termination*, 2014.
- [2] George S Boolos, John P Burgess, and Richard C Jeffrey. *Computability and Logic*. Cambridge University Press, fifth edition, 2007.
- [3] Georg Cantor. Ueber eine elementare Frage der Mannigfaltigkeitslehre. *Jahresbericht der Deutschen Mathematiker-Vereinigung*, 1:75–78, 1892.
- [4] William F Clocksin and Christopher S Mellish. *Programming in Prolog: Using the ISO Standard*. Springer, fifth edition, 2003.
- [5] Martin D Davis, Ron Sigal, and Elaine J Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. Academic Press, second edition, 1994.
- [6] Melvin Fitting. A Kripke-Kleene semantics for logic programs. *The Journal of Logic Programming*, 2(4):295–312, 1985.
- [7] Brian W Kernighan and Dennis M Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.
- [8] Elliott Mendelson. *Introduction to Mathematical Logic*. CRC Press, fifth edition, 2010.
- [9] Bertrand Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30(3):222–262, 1908.
- [10] Alfred Tarski. The semantic conception of truth: And the foundations of semantics. *Philosophy and Phenomenological Research*, 4(3):341–376, 1944.
- [11] Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937.